Faculté des Sciences et Ingénierie - Sorbonne université

Master Informatique parcours - DAC / IMA

ML - Machine Learning

Rapport de projet

Réseau de neurones DIY

Réalisé par :
SAID Faten Racha - M1 DAC
MAOUCHE Mounir - M1 IMA

Supervisé par :
Nicolas Baskiotis
Nicolas Thome

Mai 2024

Table des matières

Table des figures

Introduction

Les réseaux de neurones sont des algorithmes d'apprentissage automatique qui ont révolutionné le domaine de l'intelligence artificielle, notamment en ouvrant les portes aux *Deep Learning* qui a permis un bond considérable de performances au cours de la dernière décennie, et ce quelles que soient les tâches ou les types de données.

Le projet ici présent a pour but l'implémentation d'un réseau de neurones d'une manière polyvalente et modulaire. Un réseau de neurones est vu comme une suite de couches ou modules, que l'ont peut assembler à notre guise afin de créer un modèle pouvant apprendre d'une manière totalement automatique.

Nous allons donc étudier dans ce qui suit le processus création pas à pas d'un réseau de neurones, en partant des briques les plus simples aux architectures les plus complexes, tout cela accompagné d'expérimentations mettant en exergue l'efficacité de notre implémentation.

Chapitre 1

Principe général et module linéaire

Notre conception d'un réseau de neurones tourne autour de la notion de module. C'est pourquoi, nous commençons par définir ce que sera un module, quelles seront ses attributs, et surtout les fonctions qu'il devra comporter afin de permettre l'apprentissage et l'extension à des modèles plus complexes.

1.1 Classe abstraite *Module*

Notre code est centré sur la classe abstraite *Module*, représentation générique d'un module de réseau de neurones.

Un module peut-être un module linéaire, une fonction d'activation (sigmoïde, tangente hyperbolique...) ou encore un réseau de neurones composé lui-même de modules.

Chaque module doit répondre aux spécificités de la classe en implémentant :

— Des *paramètres* si nécessaire.

— Une fonction *forward* : Permettant de calculer la sortie du module à partir des entrées qui lui sont données.

— Des fonctions *backward* : Permettant de calculer les dérivées des sorties du module par rapport à ses entrées et ses paramètres afin de permettre la rétropropagation du gradient du coût de la fin au début du réseau.

1.2 Module *Linéaire*

Ce module effectue une simple transformation linéaire des données en les multipliant par une matrice de poids et ajoutant un biais, de type :

$$Y = W*X + b$$

avec :

— **X** : La matrice des entrées d'apprentissage, de forme (N*d), comportant en colonnes les différentes dimensions ou caractéristiques de nos exemples et en lignes les exemples d'entraînement ou de test.

— **Y** : La matrice de supervision pour chaque entrée, de forme (N*K),comportant autant de colonnes que de valeures à prédire pour une entrée.

Une convention que nous allons suivre tout au long de notre projet.

1.3 Expérimentations

1.3.1 Régression linéaire

Nous commençons par tester notre modèle sur une tâche de régression linéaire.

Données d'entraînement

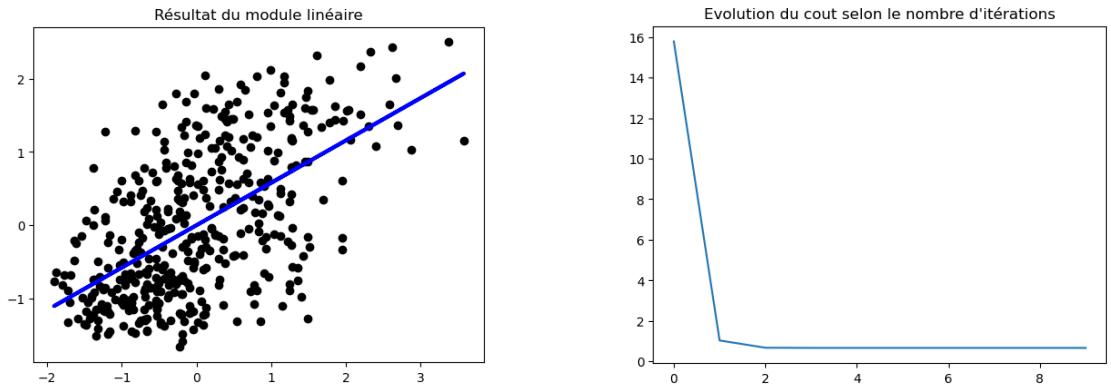


FIGURE 1.1 – Résultat et cout de la régression linéaire

Données de test

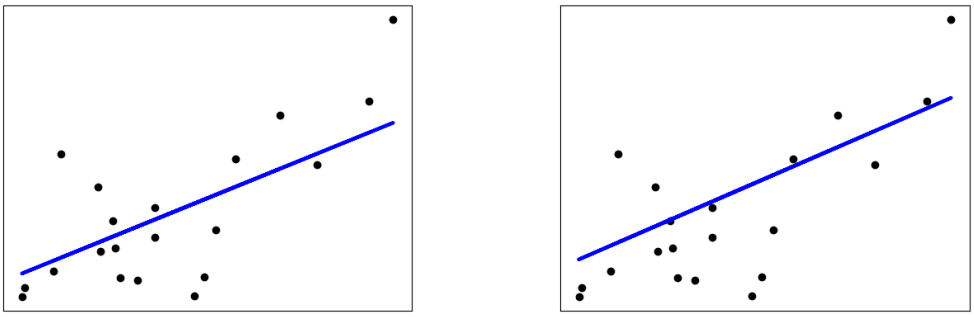


FIGURE 1.2 – Résultat en test de notre modèle (gauche) comparé avec celui du modèle linéaire de Sci-Kit learn

Nous pouvons voir que notre modèle rend des résultats convaincants et similaires à celui de la bibliothèque Sci-kitLearn. De plus le *coefficient de détermination* de notre modèle est supérieur avec un score de 0.52 contre 0.47 pour ski-kitlearn.

1.3.2 Classification binaire

Ceci est une reproduction du test basique du perceptron avec une classification de données linéairement séparables.

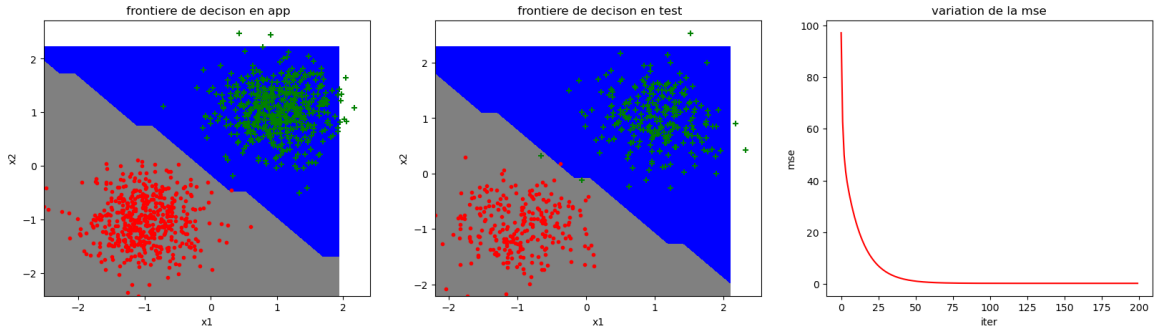


FIGURE 1.3 – Classification binaire avec notre modèle linéaire.

Chapitre 2

Modules non-linéaires

Nous introduisons maintenant des modules non-linéaires, aussi appelés *activations*, afin d'augmenter l'expressivité de nos modèles et s'adapter à de plus larges configurations de données.

2.1 Présentation

Les modules d'activation que nous avons implémentés sont :

— **Sigmoide** $\frac{1}{1+e^{-x}}$: Permet de compresser les valeurs des entrées dans un intervalle $]0;1[$. Elle est utile dans les cas où l'on cherche à extraire des probabilités à partir des données, et a l'avantage d'être dérivable, mais peut causer des problèmes de saturation lorsque les valeurs des entrées sont extrêmes. C'est pourquoi nous tâchons de **normaliser nos données** avant chaque expérimentation

— **Tangente hyperbolique** (Tanh) $\frac{1-e^{-2x}}{1+e^{-2x}}$: Cette fonction possède des propriétés similaires à la fonction sigmoide à la différence que ses valeurs sont dans l'intervalle $]-1;1[$, et symétriques par rapport à l'origine.

— **SoftMax** $\frac{e^{x_j}}{\sum_i^k e^{x_i}}$: Cette fonction est privilégiée dans les cas de régression multi-classes car elle permet de retourner une distribution de probabilités d'appartenance de la donnée à chaque classe, et permet d'ajuster ces prédiction d'une manière dépendante les unes des autres lors de la rétropropagation.

— **ReLU** $max(0, x)$: Retournant simplement la valeur de l'entrée si elle est positive (ou supérieure à un seuil selon une variante), cette fonction est particulièrement appréciée dans les réseaux de neurones profonds, car elle ne souffre pas du problème de disparition du gradient.

2.2 Expérimentations

Nous expérimentons maintenant un enchaînement de deux modules linéaires suivis respectivement d'une activation *Tanh* et *Sigmoide*, dont la rétropropagation est effectuée manuellement.

— **accuracy en train** : 0.996

— **accuracy en test** : 1.0

Ce sont des résultats concluants, notamment comparés à un réseau avec une simple couche linéaire plus activation sigmoide :

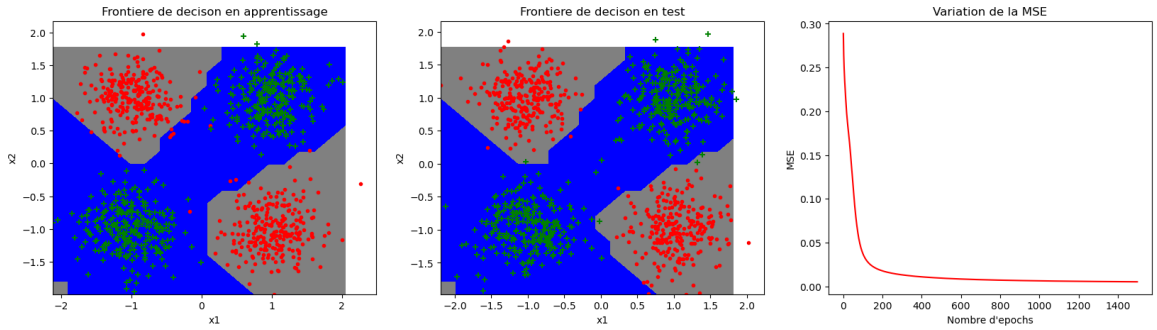


FIGURE 2.1 – Classification binaire sur des données non séparables linéairement.

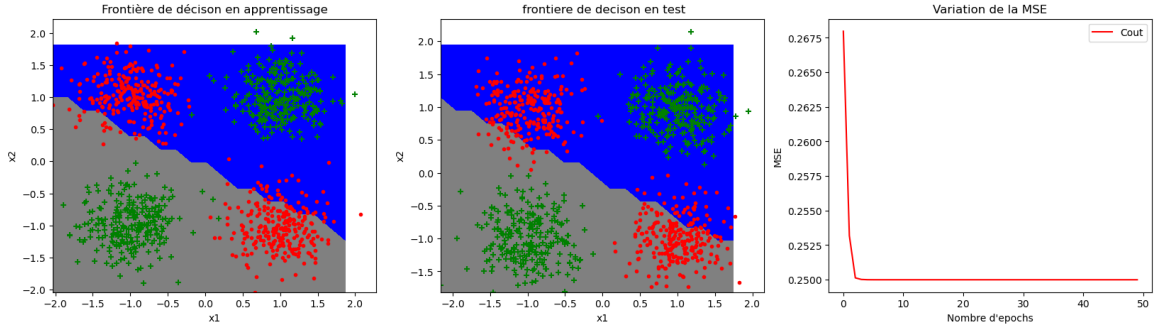


FIGURE 2.2 – Classification binaire sur des données non séparables linéairement.

Nous pouvons voir ici que le réseau n'est pas assez expressif pour apprendre la distribution des données et stagne rapidement à environs 50% d'exactitude.

— **accuracy en train** : 0.494

— **accuracy en test** : 0.511

Chapitre 3

Encapsulation

3.1 Séquentiel

Le calcul manuel des gradients étant inconcevable si l'on veut construire des réseaux de neurones profonds, nous implémentons un module *Sequentiel* qui permet de réunir une suite de modules et de les lier entre eux de sorte que la rétro-propagation puisse se faire automatiquement.

3.2 Optimiseur et descente de gradient

Ensuite, nous implémentons un *Optimiseur* qui nous permettra d'effectuer automatiquement une opération complète de mise à jour du gradient à la fois. Egalement, nous implémentons une *Descente de Gradient Stochastique (SGD)* qui découpe l'ensemble d'entraînement en batchs de taille donnée et effectue pour chaque batch une mise à jour du gradient après avoir calculé l'erreur sur ses éléments. Cette méthode permet de converger plus rapidement malgré un gradient bruité car calculé sur une partie des données et non la totalité comme auparavant.

3.3 Expérimentations

3.3.1 Mélange de 4 gaussiennes

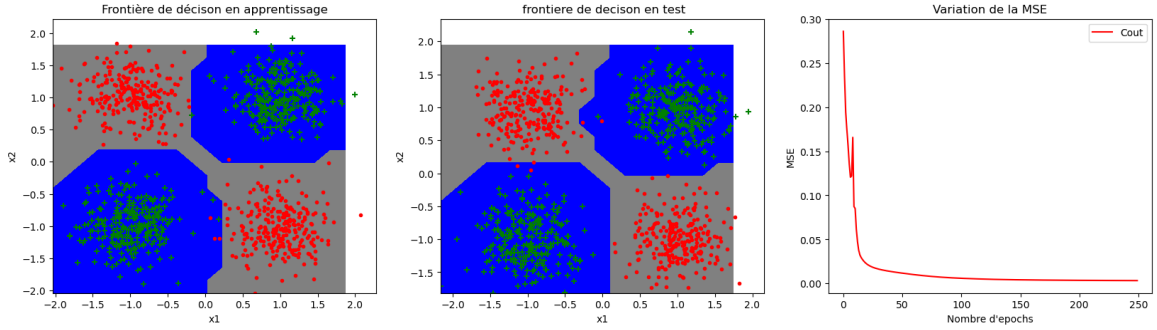


Figure 3.1 – Classification binaire sur des données non séparables linéairement.

— **accuracy en train** : 0.997

— **accuracy en test** : 0.996

Nous pouvons constater que l'ajout de la méthode de descente de gradient stochastique a porté ses fruits étant donné que le modèle a convergé plus rapidement comme le montre la stabilisation précoce du coût.

3.3.2 Variation des hyper-paramètres sur le dataset Echiquier

Nous allons dans ce qui suit utiliser un dataset plus complexe afin de mettre en évidence les impacts des différents hyperparamètres

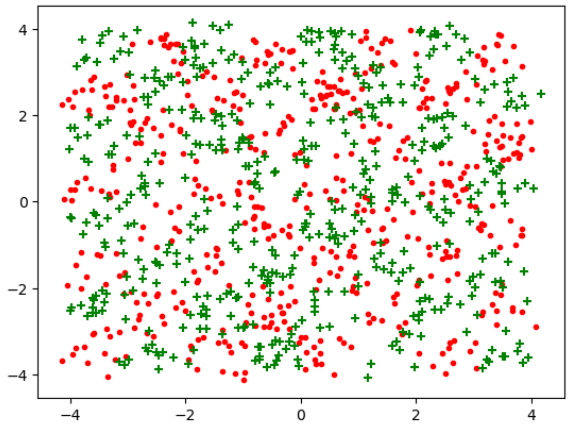


FIGURE 3.2 – Dataset echiquier.

Nombre de batchs et itérations

Nombre d'itération	Nombre de batchs	Accuracy en train	Accuracy en test
1500	10	0.906	0.782
2000	10	0.93	0.781
1000	10	0.842	0.753
600	10	0.747	0.702
600	30	0.728	0.682
600	20	0.735	0.674
600	5	0.692	0.661
300	30	0.633	0.628
300	40	0.628	0.626
600	1	0.525	0.523

TABLE 3.1 – Variation du nombre d'itérations et du nombre de batchs. Chronologie des tests : du plus petit nombre d'itérations au plus grand

Nos expérimentations on conclu que le nombre optimal de batchs est de 10, et **Notons** également que le temps d'exécution pour 40 batchs était excessivement long et ne nous a pas permis d'expérimenter avec plus d'itérations pour ce cas-là. De plus, l'augmentation du nombre d'itération permet globalement d'augmenter l'accuracy, mais dans le cas de 10 batchs, le modèle a commencé à surapprendre lorsque nous sommes arrivés à 2000 itérations.

Pas d'apprentissage

Le pas d'apprentissage détermine à quelle ampleur les paramètres sont modifiés lors de chaque mise à jour. Une valeur correcte de ce paramètre est cruciale pour l'apprentissage, car si elle est trop petite, le modèle convergera trop lentement, mais si elle est trop grande, le modèle finira par diverger et ne jamais trouver l'optimum.

Nous cherchons à présent le pas d'apprentissage optimal pour la configuration qui a rendu les meilleurs résultats lors du test précédent : 1500 itérations et 10 batchs :

Pas d'apprentissage	Accuracy en train	Accuracy en test
1e-02	0.906	0.782
1e-03	0.596	0.586
1e-01	0,5	0,485

TABLE 3.2 – Variation du pas d'apprentissage.

Visualisons maintenant le résultat du modèle optimal sur notre échiquier :

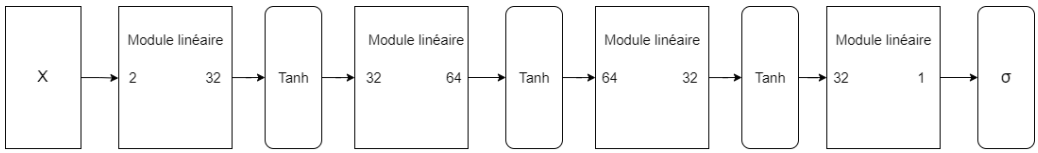


FIGURE 3.3 – Architecture du réseau de neurones à 4 couches utilisé.

Nombre d'itérations	Pas d'apprentissage	Nombre de batchs
1500	1e-2	10

TABLE 3.3 – Hyper-paramètres du réseau.

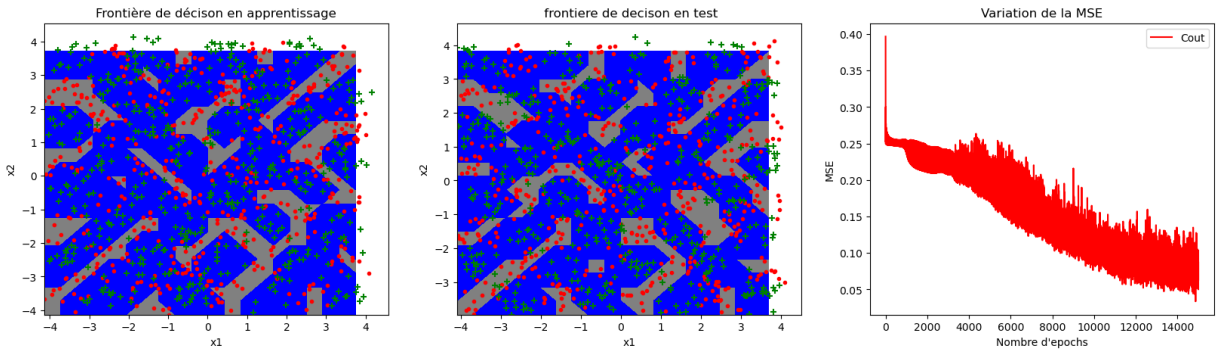


FIGURE 3.4 – Visualisation du résultat de la meilleure configuration sur le dataset de l'échiquier.

— **accuracy en train** : 0.906

— **accuracy en test** : 0.782

3.3.3 Expérimentations supplémentaires

— **Réseau de neurones à 2 couches** : Accuracy ne dépassant pas les 60% quelle que soit la dimension des neurones cachés. Cette architecture, même si elle a permis d'apprendre des données non-linéaires lors des tests sur les 4 gaussiennes, n'est pas assez expressive pour des données aussi complexes que celles de l'échiquier.

— **Permutation des données** : Ce paramètre n'influe pas sur les performances, car les données de l'échiquier paraissent déjà distribuées d'une manière indépendante les unes des autres.

— **Changement de la fonction d'activation** : La fonction *Tanh* accompagnée d'une *sigmoide* est celle qui a rendu les meilleurs résultats, devant la *ReLU*, pour ces tests précis.

— **Divergence du modèle** Nous avons rencontré une divergence lors d'une mise trop élevée (1e-1) du pas d'apprentissage, illustrée par cette oscillation de la valeur du coût

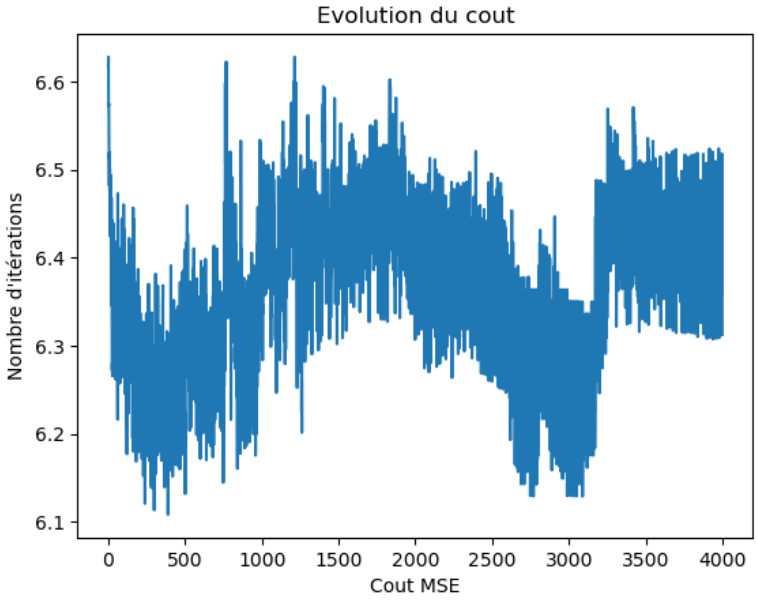


FIGURE 3.5 – Non-convergence suite à un pas de gradient trop élevé

Chapitre 4

Multi-classes

Notre expérimentation portera sur la **classification d'images** de nombres manuscrits. Celle-ci diffère des précédentes par le fait que la nouvelle tâche est de prédire, pour chaque image, une classe parmi 10 possible, contrairement à 2 précédemment.

4.1 Présentation

Nous modélisons cet aspect multi-classes des étiquettes à l'aide d'un vecteur one-hot encodé pour chaque classe. Le modèle devra ensuite retourner une probabilité d'appartenance à chaque classe avec la fonction *SoftMax*, qui seront transmises à une fonction de coût d'*entropie croisée*.

$$CE(y,\hat{\mathrm{Y}}) = -\hat{\mathrm{Y}}_y$$

Avec $\hat{\mathrm{Y}}_y$ étant la probabilité retournée pour la classe à prédire.

Celle-ci est préférée à la MSE (somme des moindres carrés) que nous avons jusque-là utilisée car son gradient est élevé aux alentours de 0. Elle permet donc de mieux pénaliser les cas où le modèle retourne une probabilité proche de zéro pour la classe qu'il est censé prédire.

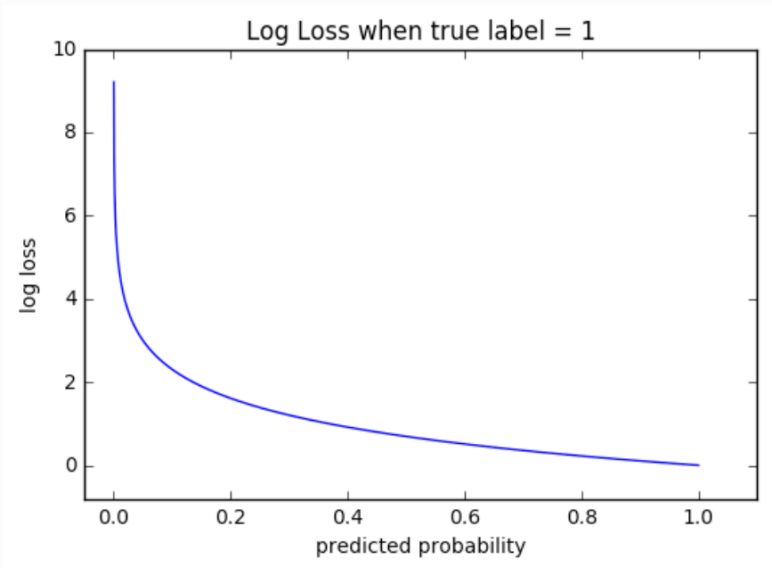


Figure 4.1 – Comportement de la CrossEntropy Loss en fonction de la probabilité prédite de la bonne classe.

Notons que la formule ci-dessus est une forme simplifiée, du fait de la binarité de l'appartenance aux classes : une image appartient à une seule classe et à aucune des autres.

4.2 Expérimentations

Les résultats de notre expérimentation ont abouti à un module à 2 couches linéaires :

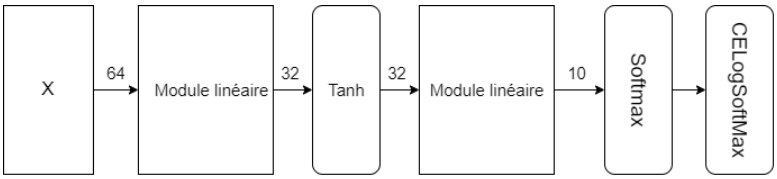


FIGURE 4.2 – Réseau de neurones optimal selon les expérimentations pour la classification multi-classes

La meilleure configuration d'hyper-paramètres est la suivante :

Nombre d'itérations	Pas d'apprentissage	Nombre de batchs	Permutation des données
150	1e-2	20	Non

TABLE 4.1 – Hyper-paramètres optimaux du réseau.

En ce qui concerne la permutation des données d'entraînement, le fait de la désactiver a donné une accuracy a diminué la variance des modèles, de sorte que l'accuracy en train a baissé mais celle en test a augmenté. Nous nous attendions à observer cela avec l'activation de ce paramètre étant donné qu'il est censé permettre une meilleure généralisation en supprimant l'effet de l'ordre des données d'entraînement.

— **accuracy en train** : 0.919

— **accuracy en test** : 0.892

Evolution du coût :

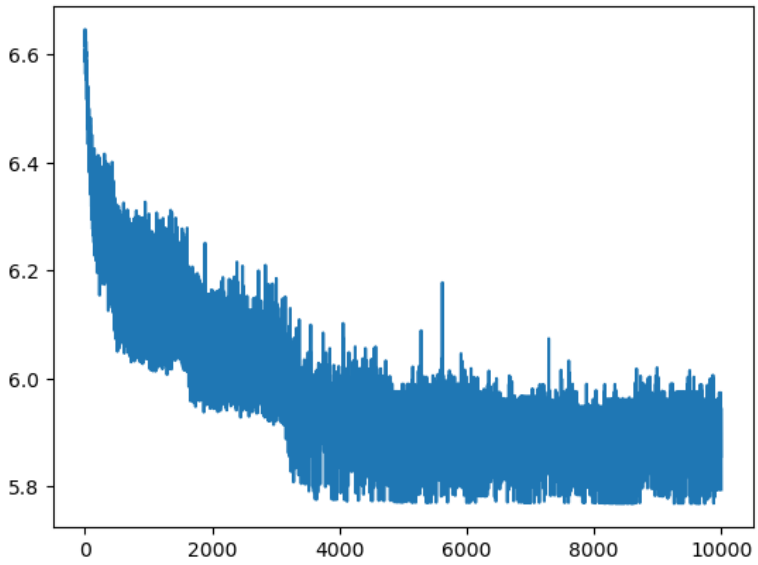


FIGURE 4.3 – Evolution de la Cross-Entropy loss

Prédictions

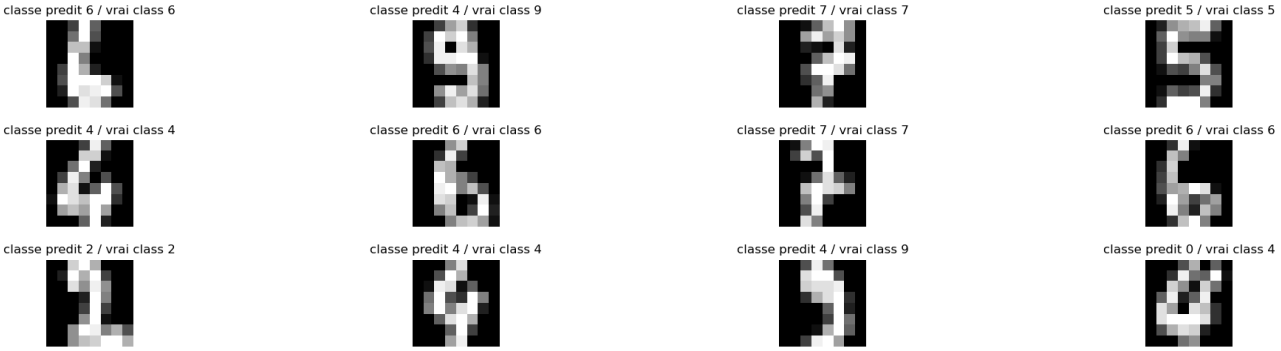


FIGURE 4.4 – Exemples de prédictions du modèle comparées aux vrais labels

La figure 4.4 illustre un certains nombre d'instances de test sur lesquels notre modèle a été évalué. Nous pouvons globalement observer les très bonne performances de ce derniers même sur des images assez peu évidente comme la deuxième image de la troisième ligne.

Chapitre 5

Auto-Encodeur

Un auto-encodeur est une architecture pour l'apprentissage d'un encodage des données dans un espace différent, appelé *latent*, et le plus souvent de dimension inférieure à l'espace de départ.

5.1 Présentation

Son fonctionnement est en deux parties :

— L'apprentissage des représentations avec un *encodeurs*

— La reconstruction des données originales à partir des représentations

Ainsi, l'autoencodeur est un type d'algorithme non-supervisé qui permet de multiples opérations sur les données telles la compression, le débruitage, la segmentation, ou l'apprentissage de caractéristiques cachées dans des données.

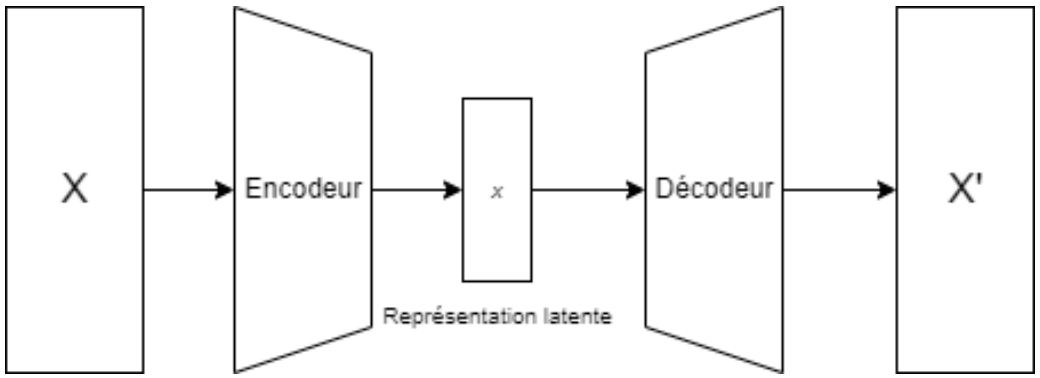


Figure 5.1 – Architecture typique d'un auto-encodeur.

5.2 Expérimentations

5.2.1 Reconstruction

Notre première tâche consiste à reconstruire les images du dataset *USPS* après les avoir normalisées et encodées.

Les résultats sont assez concluants avec certaines images reconstruites avec une grande précision, mais également quelques erreurs.

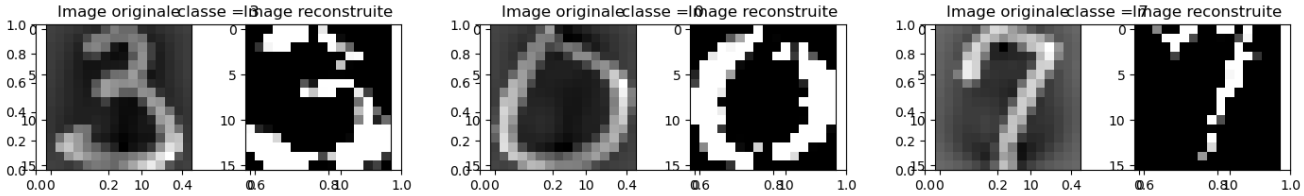


FIGURE 5.2 – Exemples de bonne reconstruction par notre auto-encodeur

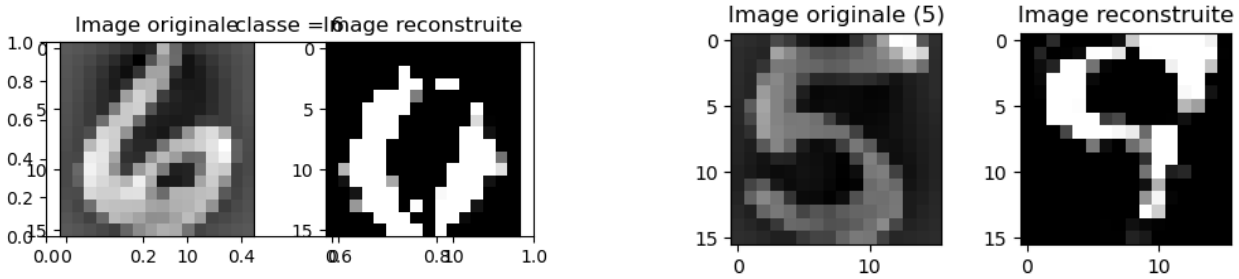


FIGURE 5.3 – Exemples d'erreurs de reconstruction de l'auto-encodeur

5.2.2 Clustering

Comparons le clusering des images induit par l'espace latent avec celui de l'espace de départ en utilisant un k-means.

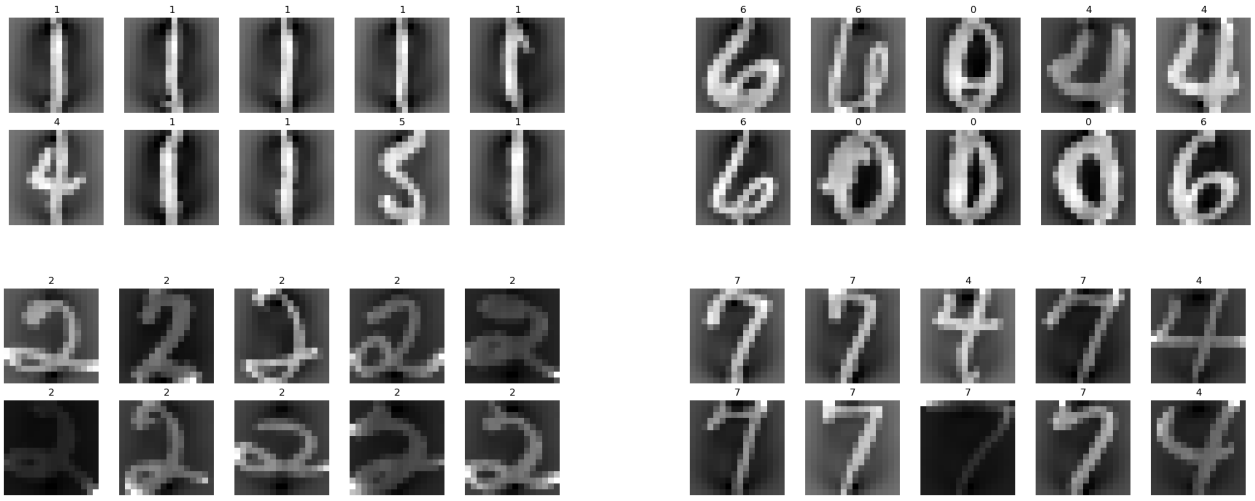
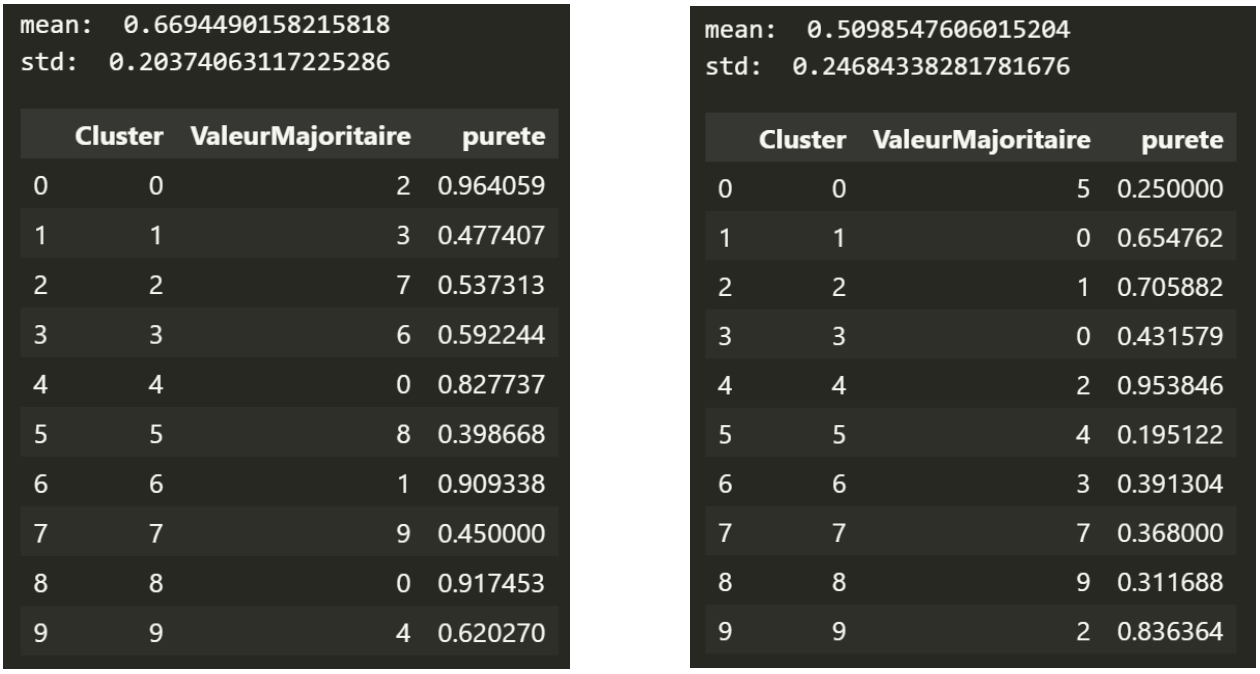


FIGURE 5.4 – Clustering induit par l'espace latent

Résultats et évaluation

Métrique	Clustering espace original	Clustering espace latent
Score silhouette	0.094	0.398
Inertie	172488.64	2653

TABLE 5.1 – Métriques des clusterings.



mean: 0.6694490158215818
std: 0.20374063117225286

	Cluster	ValeurMajoritaire	purete
0	0	2	0.964059
1	1	3	0.477407
2	2	7	0.537313
3	3	6	0.592244
4	4	0	0.827737
5	5	8	0.398668
6	6	1	0.909338
7	7	9	0.450000
8	8	0	0.917453
9	9	4	0.620270

mean: 0.5098547606015204
std: 0.24684338281781676

	Cluster	ValeurMajoritaire	purete
0	0	5	0.250000
1	1	0	0.654762
2	2	1	0.705882
3	3	0	0.431579
4	4	2	0.953846
5	5	4	0.195122
6	6	3	0.391304
7	7	7	0.368000
8	8	9	0.311688
9	9	2	0.836364

FIGURE 5.5 – Puretés des clusterings sur les données originales (gauche) et les représentations latentes (droite)

Pour calculer la *pureté*, nous avons pris l'image le chiffre le plus présent dan chaque cluster et avons calculé sa proportion dans le cluster.

Nous pouvons dire que le clustering des images originales paraît meilleur que celui de l'espace latent. Nous tenons à préciser que nos expérimentations avec les auto-encodeurs étaient limitées en terme de profondeur de neurones et d'epochs à cause de contraintes matérielles.

T-SNE

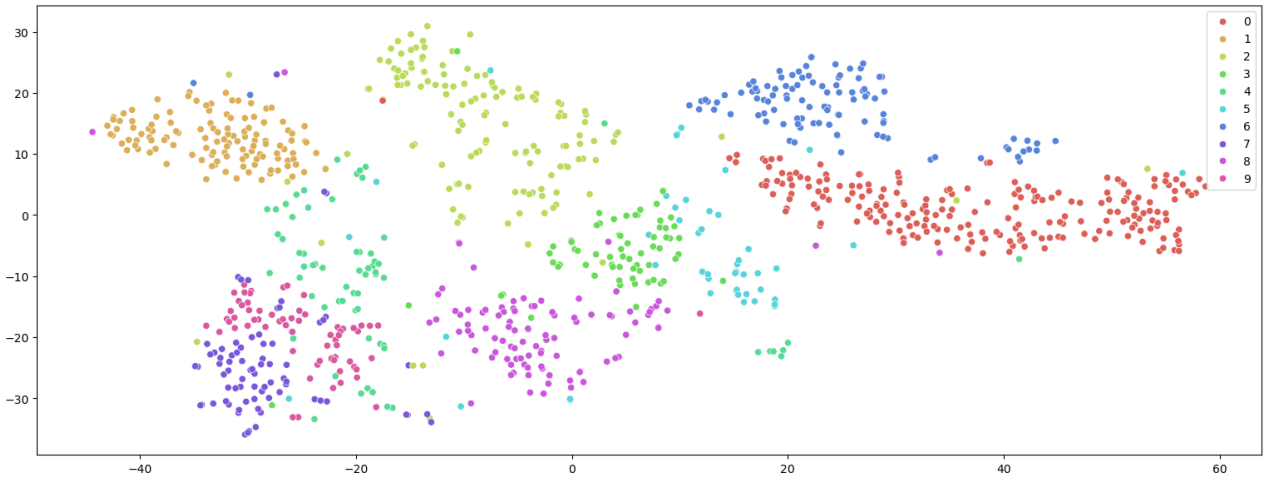


FIGURE 5.6 – Projection de la projection t-SNE des images originales

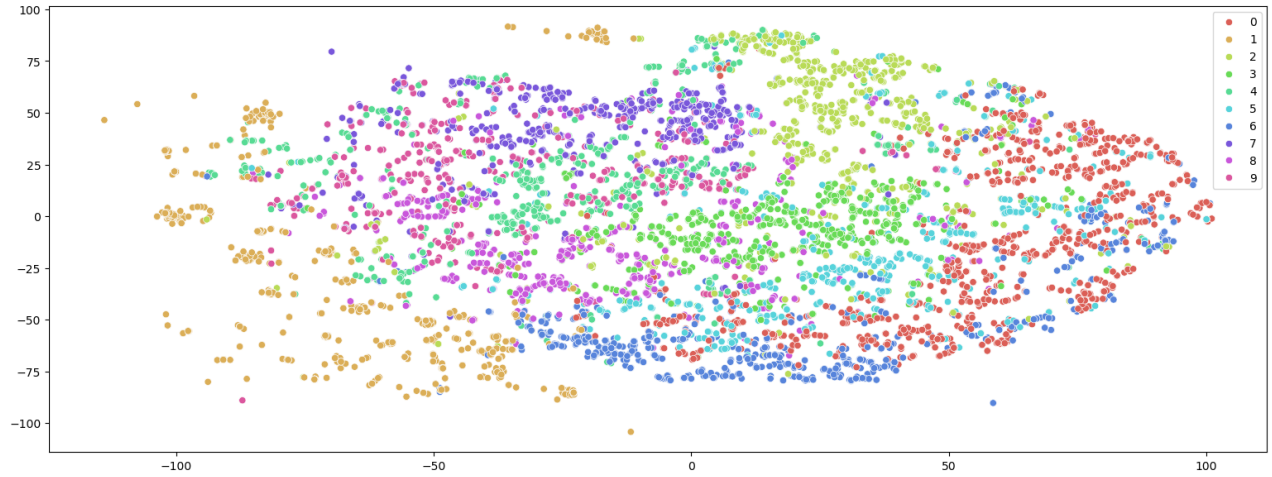


FIGURE 5.7 – Projection t-SNE des représentations latentes

5.2.3 Classification

Maintenant, nous récupérons les représentations latentes apprises par l'autoencodeur et les utilisons pour apprendre à classifier les images.

Tailles couches cachées	Fonctions d'activation	Fonction de coût	Accuracy train	Accuracy test
32	Tanh et SoftMax	CELogSoftMax	0.647	0.634
32	ReLU et SoftMax	CELogSoftMax	0.489	0.515

TABLE 5.2 – Classification sur les représentations latentes.

Nous avons de meilleurs résultats avec la fonction Tanh en tant qu'activation et une couche cachée de 32 neurones.

Chapitre 6

Convolution

Les réseaux neuronaux convolutionnels, couramment appelés CNN (Convolutional Neural Networks) ou ConvNet, représentent une classe de réseaux de neurones spécifiquement conçus pour le traitement des images numériques. Ces réseaux sont particulièrement efficaces pour des tâches détection d'objets et d'apprentissage de caractéristiques, grâce à une architecture composée de couches distinctes et spécialisées.

6.1 Présentation

Une image numérique peut être décrite comme une matrice de pixels, où chaque pixel est représenté par une valeur d'intensité binaire (noire et blanc) ou les coordonnées de la couleur dans un espace défini (RGB, HSV etc.). Les CNN exploitent cette structure en grille pour extraire et hiérarchiser des caractéristiques visuelles à travers plusieurs couches de traitement.

Un réseau convolutionnel comprend trois types de couches principales :

— **Couche convolutionnelle** : Cette couche applique des filtres ou kernel à l'image d'entrée, permettant d'extraire des caractéristiques locales telles que les bords, les textures et les motifs. Chaque opération de convolution est une multiplication matricielle entre un filtre et une région de la matrice d'entrée. Ces opérations permettent de capturer les propriétés locales de l'image, en se concentrant sur des motifs spécifiques présents dans différentes régions.

— **Couche de pooling :** Cette couche a pour but de réduire la dimensionnalité des représentations intermédiaires, tout en conservant les informations essentielles. Les techniques de pooling couramment utilisées incluent le max-pooling et l'average-pooling, qui sélectionnent respectivement la valeur maximale ou moyenne dans une région de la matrice. Cette réduction diminue le nombre de paramètres et la complexité computationnelle du réseau, tout en aidant à contrôler le surapprentissage.

— **Couche entièrement connectée :** Cette couche finale est similaire à un réseaux de neurones classiques. Chaque neurone y est connecté à tous les neurones de la couche précédente, ce qui permet d'intégrer et de combiner les caractéristiques extraites par les couches précédentes pour effectuer des prédictions finales. Cette couche est généralement utilisée pour la classification des données en fonction des caractéristiques apprises.

Reférence de la figure 6.1 : developersbreach.com/convolution-neural-network-deep-learning

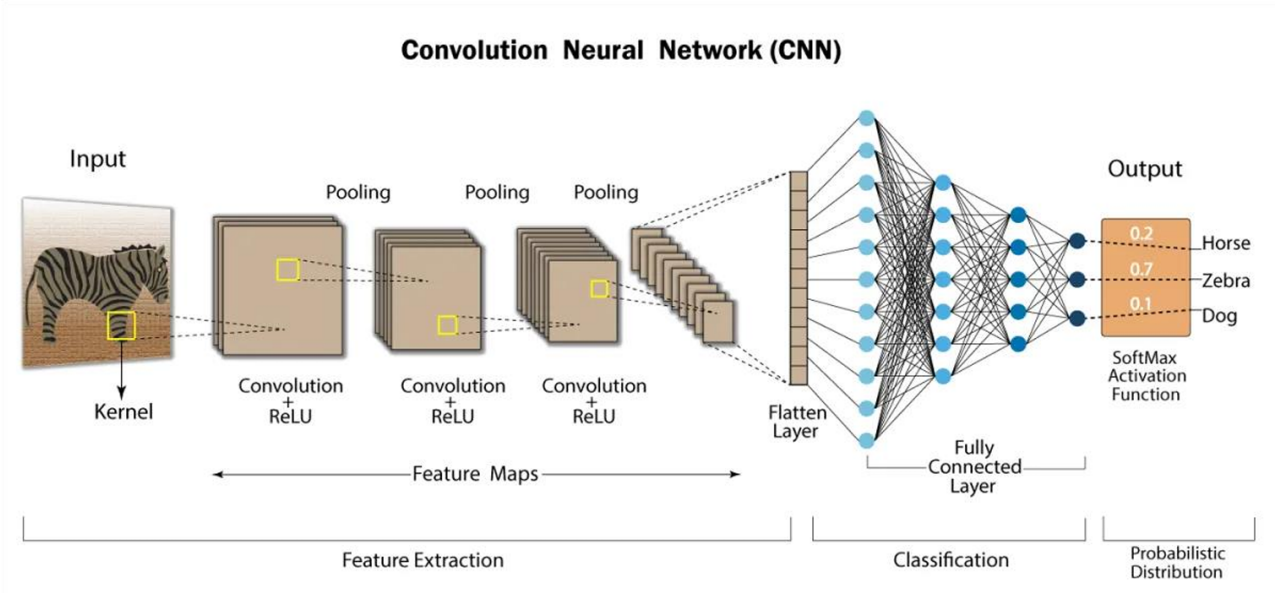


FIGURE 6.1 – Architecture d'un réseau de neurones convolutif

6.2 Expérimentations

Dans le cadre de ce projet, nous présentons les résultats de l'implémentation des modules Conv1D et Conv2D ; la différence principale étant dans la dimension du noyaux de convolution le premier étant en 1D et le second en 2D. Il est important de noter que ces modules convolutionnels nécessitent un temps de calcul significatif en raison des nombreuses opérations de multiplication qu'ils impliquent. Ainsi, l'utilisation d'unités de traitement graphique (GPU) est souvent indispensable pour accélérer ces calculs et améliorer l'efficacité des implémentations des réseaux convolutionnels.

6.2.1 Convolution 1D

Pour cette approche, nous avons choisis d'appliquer le module Conv1D sur les données USPS.

L'architecture du réseau choisi est la suivante :

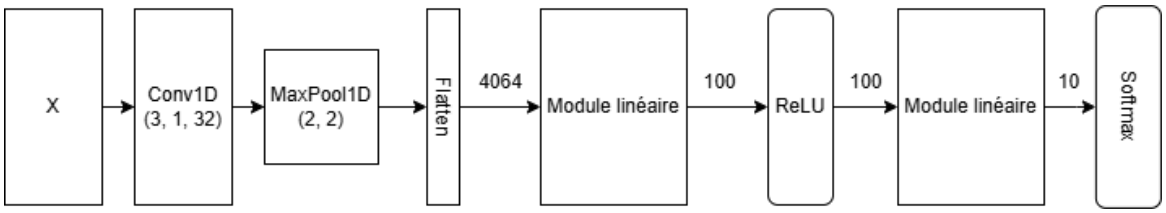


FIGURE 6.2 – Architecture du modèle Conv1D

Les performances obtenues avec ce modèle sont assez bonnes compte tenu du peu de ressources à notre disposition pour effectuer des tests. Les scores en *accuracy* train et en test

obtenus respectivement de 0.574 et 0.548, pour 100 *epochs* et 1100 images en train et 500 en test.

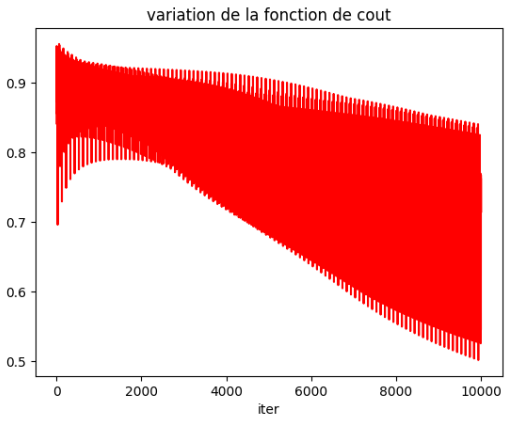


FIGURE 6.3 – Evolution de la CrossEntropy Loss en fonction du nombre d'itérations - Conv1D

La Figure 6.3 illustre l'évolution du coût au fur et à mesure des itérations. Nous observons une diminution de ces valeurs, ce qui laisse penser qu'avec plus de ressources matérielles qui nous permettraient d'étendre nos expérimentations à davantage d'itérations et un plus grand ensemble d'apprentissage/test, nous pourrions atteindre une convergence du modèle.

6.2.2 Convolution 2D

Pour cette approche, nous avons choisis d'appliquer le module Conv2D sur les données USPS.

L'architecture du réseau choisi est la suivante :

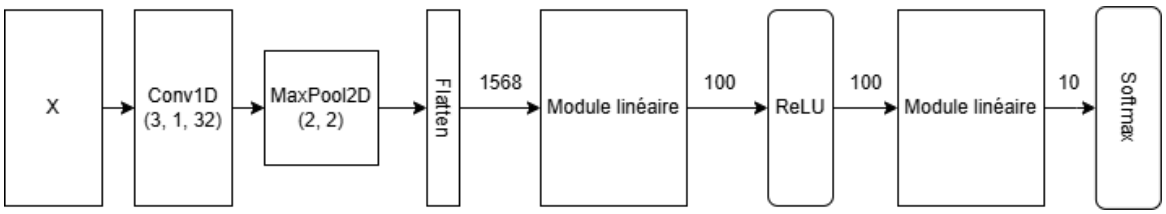


FIGURE 6.4 – Architecture du modèle Conv2D

Les performances obtenues avec ce modèle sont assez bonnes compte tenu du peu de ressources à notre disposition pour effectuer des tests. Les scores en *accuracy* train et en test obtenus respectivement de 0.58 et 0.505, pour 165 *epochs* et 1000 images en train et 200 en test.

La Figure 6.5 illustre l'évolution du coût au fur et à mesure des itérations. De même que pour l'analyse précédente, nous observons une diminution de ces valeurs, ce qui laisse penser qu'avec plus de ressources matérielles qui nous permettraient d'étendre nos expérimentations à davantage d'itérations et un plus grand ensemble d'apprentissage/test, nous pourrions atteindre une convergence du modèle.

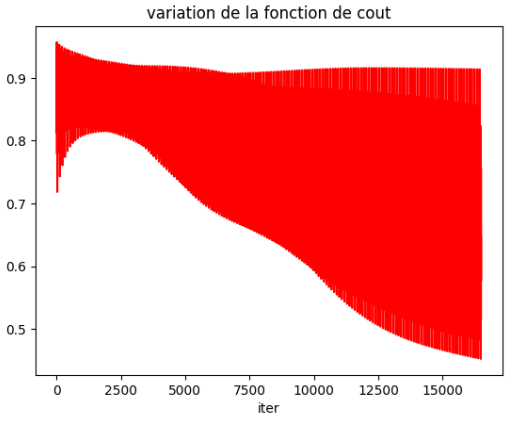


FIGURE 6.5 – Evolution de la CrossEntropy Loss en fonction du nombre d'époch s - Conv2D

Pour avoir ces résultats nous avons procédé à une série d'expérimentations que nous décrivons par le tableau 6.1.

Nombre de batchs	Nombre d'epochs	Pas d'apprentissage	Accuracy train	Accuracy test
1	150	1e-2	0.54	0.475
1	160	1e-1	0.571	0.5
50	160	1e-1	0.571	0.5
100	165	1e-1	**0.58**	**0.505**

TABLE 6.1 – Classification sur les représentations latentes.

Nous remarquons que plus le nombre de batchs augmente, plus les perfomances en train et en test augmtent également ; ce qui montre que le modèle apprend mieux lorsqu'il est entraîné sur de plus petits ensembles de données car cela implique plus de mises à jours des poids du modèle. Notons que c'est ce dernier point qui manque justement à notre modèle pour qu'il performe mieux, sauf que nous n'avons pas assez de ressources matérielles (RAM) pour supporter plus d'itérations ; soit plus de mises à jour des poids.

Conclusion générale

Dans ce projet, nous avons implémenté et testé divers modules d'un réseau neuronal qui s'inspirent d'anciennes implémentations de la bibliothèque PyTorch.

Nous avons commencé par les composantes élémentaires qui permettent de créer des réseaux simples que nous avons fait évoluer vers des architectures plus complexes qui permettent d'appréhender diverses tâches de *Machine Learning* telles que la régression,la classification, la reconstruction de données, la segmentation etc.

Ce projet nous a permis de comprendre les rouages des réseaux de neurones en expérimentant les différentes méthodes d'apprentissage et d'évaluation applicables. Ces expérimentations ont été mises en exergue par notre multitude de tests impliquant des modèles tels que les autoencodeurs ou encore les CNN et dont les résultats étaient satisfaisants quant à la qualité de notre implémentation.